Git

Introduction:

"Git allows you to save the state of your project files, called snapshots or commits"

- As each programmer commits their changes,
  - other programmers can pull these updates onto their computers.

- The version control system tracks what commits were made,
  - who made them, and when they made them,
  - along with the developers' comments describing the changes.

- Version control manages a project's source code in a folder called a repository, or repo.

- you should keep a separate Git repo for each project you're working on

- [branching and merging] --- that help programmers collaborate:
  - needs to discuss in an other tutorial

Installing Git:

- Git might already be installed on your computer:
  - to find out, run

    git --version

- If you see a "command not found" error message,
  - you must install Git [linux], run

    sudo apt install git-all

Configuring Your Git Username and Email:

- After installing Git, you need to configure your name and email so your commits include your author information,
    - from a terminal, run the following *git config* commands,

        git config --global user.name “Alex”
        git config --global user.email abc@gmail.com

- you can change it by running the **git config** command

- list the current Git configuration settings using the **git config --list** command.

The Git Workflow:

Using a Git repo involves the following steps.

- *First*, you create the Git repo by running the **git init** or **git clone** command

- *Second*, you add files with the **git add <filename>** command for the repo to track

- *Third*, once you’ve added files, you can commit them with the **git commit -am "<descriptive commit message>"**

Note: You can view the help file for each of these commands by running git help <command>, such as **git help init** or **git help add**

How Git Keeps Track of File Status:

All files in a working directory are either tracked or untracked by Git.

- Tracked files are the files that have been *added and committed to the repo*, whereas every other file is *untracked*.

- to the Git repo, untracked files in the working copy might as well not exist.

- the tracked files exist in one of three other states:
    - The *committed state* is when a file in the working copy is identical to the repo's most recent commit. (This is also sometimes called the unmodified state or clean state.)

    - The *modified state* is when a file in the working copy is different from the repo's most recent commit.

    - The *staged state* is when a file has been modified and marked to be included in the next commit. We say that the file is staged or in the staging area. (The staging area is also known as the *index or cache*.)

You can add an untracked file to the Git repo, in which case it becomes tracked and staged.

- You can then *commit* staged files to put them into the committed state.

- You don't need any Git command to put a file into the modified state; once you make changes to a committed file, it's automatically labeled as modified.

At any step after you've created the repo, run **git status** to view the current status of the repo and its files' states.

Creating a Git Repo on Your Computer:

Git is a distributed version control system, which means it stores all of its snapshots and repo metadata locally on your computer in a folder named .git.

- From a terminal, run the following commands to create the .git folder.:
  mkdir <foldername>
  cd <foldername>
  git init

Note:
- When you convert a folder into a Git repo by running git init , all the files in it start as untracked.
- The presence of this .git folder makes a folder a Git repository; without it, you simply have a collection of source code files in an ordinary folder.

- *RUNNING GIT STATUS WITH THE WATCH COMMAND*
  - *https://inventwithpython.com/watch.exe -- download and place this file into a path folder [on linux already installed]*
    watch "git status"
    [we can use this command to see git status every 2 seconds]

  - you can use the following command to see the summary of commits, *[updated in real time]*
    watch "git log –online"

Adding Files for Git to Track:

Only tracked files can be committed, rolled back, or otherwise interacted with through the git command.

- run **git status** to see the status of the files in the project folder

Committing Changes:

After adding new files to the repo, you can continue writing code for your project.

- When you want to create another snapshot:
  - you can run **git add .** to stage all modified files
  - **git commit –m <commit message>** to commit all staged files
  - we can do with the single **git commit –am <commit message>** command
  - instead of every modified file, you can omit the –a option from –am and specify the files after the commit message, such as **git commit –m <commit message> file1.py file2.py**
  - if you forget to add the -m "<message>" command line argument,
    - Git will open the Vim text editor in the Terminal window *[press ESC key and enter qa! to safely exit Vim and cancel commit]*
  - By committing the *staged files*, you've moved them back to the *committed state*

Summary: when we added files to the Git repo, the files went from untracked to staged and then to committed.

Note: you can't commit folders to a Git repo. Git automatically includes folders in the repo when a file in them is committed, but you can't commit an empty folder.

- If you made a typo in the most recent commit message, you can rewrite it using the **git commit --amend -m "<new commit message>"**

Using git diff to View Changes Before Committing:

Before you commit code, you should quickly review the changes you'll commit when you run git commit:

- You can view the differences between the code currently in your working copy and the code in the latest commit using the **git diff** command

How Often Should I Commit Changes?

You should commit code when you've completed an entire piece of functionality, such as a feature, class, or bug fix.

- Don't commit any code that contains syntax errors or is obviously broken

- Commits can consist of a few lines of changed code or several hundred, but either way,

- you should be able to jump back to any earlier commit and still have a working program

- You should always run any unit tests before committing

- Ideally, all your tests should pass (and if they don't pass, mention this in the commit message).

  Deleting Files from the Repo:

  If you no longer need Git to track a file, you can't simply delete the file from the filesystem.

  - you must delete it through Git using the **git rm** command which also tells Git to untrack the file

  - the git rm command stages the file. You need to commit file deletion just like any other change **git commit -m "Deleting"**

Renaming and Moving Files in the Repo:

Similar to deleting a file, you shouldn't rename or move a file in a repo unless you use Git.

- use the **git mv** command, followed by **git commit** .

- You can also rename and move a file at the same time by passing **git mv** a new name and location.

Viewing the Commit Log:

The **git log** command outputs a list of all commits:

- If you want to set your files to a commit that's earlier than the latest one,
  - you need to find the commit hash, a 40-character string of hexadecimal digits (composed of numbers and letters A to F)

- it's common to use only the first seven digits like, *962a8ba*

- To display the contents of a file as it was at a particular commit, you can run the **git show <hash>:<filename>** command

Recovering Old Changes:

Undo with:

- **git revert <SHA>**

Undoing Uncommitted Local Changes:

If you've made uncommitted changes to a file but want to revert it to the version in the latest commit,

- you can run **git restore <filename>**

- You can also run **git checkout .** to revert all changes you've made to every file in your working copy

Unstaging a Staged File:

If you've staged a modified file by running the git add command on it but now want to remove it from staging:

- run **git restore --staged <filename>**

Rolling Back the Most Recent Commits:

- run **git revert -n HEAD~3..HEAD** *[to move to a point in previous 3 commits]*

- then **run git add .** and **git commit -m "<commit message>"**

Rolling Back to a Specific Commit for a Single File:

- use the **git show <hash>:<filename>** command to display this file as it was after a specific commit.

- Using the **git checkout <hash> -- <filename>**

- use commands **[git add and git commit]** to stage and commit these changes

GitHub and the git push Command:

Git repos can exist entirely on your computer, many free websites can host clones of your repo online:

- letting others easily download and contribute to your projects

- the clone also acts as an effective backup.

- **Pushing an Existing Repository to GitHub:**

    - git command adds GitHub as a remote repo corresponding to your local repo:
      git remote add origin <github repo address/link>

    - You then push any commits you've made on your local repo to the remote repo using the:
      git push -u origin master

- After this first push, you can push all future commits from your local repo by simply running:

  git push

Note: Pushing your commits to GitHub after every commit is a good idea to ensure the remote repo on GitHub is up to date with your local repo

Cloning a Repo from an Existing GitHub Repo:

Create a new repo on the GitHub website, select the Initialize this repository with a README checkbox.

To clone this repo to your local computer:

- use your repo's URL with the **git clone <url>** command to download it to your computer

git clone command is also useful in case your local repo gets into a state that you don't know how to undo;

- delete the local repo, and use **git clone** to re-create the repo